IV. Allgemeine Entwurfs- und Optimierung

- Greedy-Algorithmen: die Lösung wird Schritt für Schritt verbessert, zu jedem Zeitpunkt wird die lokal optimale Verbesserung durchgeführt. **16**
- Dynamische Programmierung: die optimale Lösung wird aus rekursiv gewonnenen optimalen Lösungen für Teilprobleme berechnet. Der Zeitaufwand bei der Rekursion wird dadurch verbessert, dass Ergebnisse für die Teillösungen tabelliert werden und ggf. aus der Tabelle genommen werden, statt neu berechnet zu werden. **15**
- Amortisierte Analyse: Erlaubt es, den Aufwand für eine ganze Folge von Operationen besser abzuschätzen, als durch einfaches Aufsummieren. Sinnvoll, wenn es teure Operationen gibt, die sich durch eine “Verbesserung” der Datenstruktur “amortisieren”. **17** Anwendung: *union find* Datenstruktur. **21**
- Backtracking: Man exploriert systematisch (Tiefensuche) alle Möglichkeiten, verwirft aber einen Teilbaum, sofern festgestellt werden kann, dass dieser keine (optimale) Lösung enthalten kann.

Greedy-Algorithmen

Entwurfsprinzip für Algorithmen, speziell für Optimierungsprobleme.

- Grundsätzlich anwendbar, wenn Lösung eines Problems sich als Folge von Einzelentscheidungen / partiellen Lösungen ergibt.

Beispiel: Finden einer Fahrtroute: Einzelentscheidungen = jeweils nächstes Ziel.

- Lösung L wird als Folge von partiellen Lösungen $L_0, L_1, L_2, \dots, L_n = L$ konstruiert. Die partielle Lösung L_{i+1} ergibt sich aus L_i , indem unter allen Ein-Schritt-Erweiterungen von L_i diejenige gewählt wird, die den Nutzen oder Gewinn maximiert. Als Anfang L_0 nimmt man meist die leere partielle Lösung.
- Ist die optimale Lösung nur auf dem Umweg über schlechte partielle Lösungen erreichbar, so verfehlt diese Heuristik das Ziel
- Ein Greedy Algorithmus liegt vor, wenn diese lokal optimierende greedy Strategie tatsächlich zur beweisbar besten Lösung führt.

Beispiel: ein Auswahlproblem

Gegeben: Menge $A = \{(s_1, e_1), \dots, (s_n, e_n)\}$ mit $s_i \leq e_i \in \mathbb{R}^+$ für alle i .

Ziel: Finde $B \subseteq \{1, \dots, n\}$ maximaler Größe mit $s_i \geq e_j$ oder $s_j \geq e_i$ für alle $i \neq j \in B$.

Annahme: A nach den e_i sortiert: $e_1 \leq e_2 \leq \dots \leq e_n$.

Greedy-Algorithmus:

```
 $B \leftarrow \emptyset$   
 $m \leftarrow 0$   
for  $i \leftarrow 1$  to  $n$  do  
    if  $s_i \geq m$   
        then  $B \leftarrow B \cup \{i\}$   
             $m \leftarrow e_i$   
return  $B$ 
```

Satz: Der Greedy-Algorithmus findet hier stets eine optimale Lösung für das Auswahlproblem.

Korrektheitsbeweis

Dass solch ein Greedy Algorithmus korrekt ist, kann man (wenn es überhaupt der Fall ist!) mit der folgenden Strategie zu beweisen versuchen.

- Versuche als Invariante zu zeigen: L_i lässt sich zu einer optimalen Lösung erweitern. Dies gilt sicher am Anfang wo noch keine Entscheidungen getroffen worden sind. Wichtig ist also, dies beim Übergang von L_i auf L_{i+1} zu erhalten. Dabei hilft das folgende Austauschprinzip.
- Sei L_i eine partielle Lösung und L eine sie erweiternde optimale Lösung. Sei L_{i+1} diejenige partielle Teillösung, die vom Greedy Algorithmus ausgewählt wird. Dann lässt sich L zu einer optimalen Lösung L' umbauen, die dann auch L_{i+1} erweitert.

Anwendung im Beispiel

Die triviale Lösung $B = \emptyset$ lässt sich zu einer optimalen Lösung erweitern (nur kennen wir sie nicht...).

Sei B_i zu optimaler Lösung B erweiterbar. Nehmen wir an, diese optimale Lösung enthalte nicht das vom Greedy Algorithmus gewählte Intervall (s_j, e_j) . Sei (s, e) das Intervall in $B \setminus B_i$ mit dem kleinsten Endzeitpunkt e . Ersetzen wir es durch die vom Greedy Algorithmus getroffene Wahl, so liegt immer noch eine optimale Lösung vor.

Klassische Anwendung: Steinitz'scher Austauschsatz

Satz: Sei $v_1, \dots, v_k$ eine Menge linear unabhängiger Vektoren und B eine Basis, die $\{v_1, \dots, v_k\}$ erweitert. Sei $\{v_1, \dots, v_k, u\}$ linear unabhängig. Dann existiert ein Vektor $u' \in B \setminus \{v_1, \dots, v_k\}$ sodass $B \setminus \{u'\} \cup \{u\}$ auch eine Basis ist.

Beweis: Sei $B = \{v_1, \dots, v_k, w_1, \dots, w_l\}$. Repräsentiere u in der Basis B als $u = \lambda_1 v_1 + \dots + \lambda_k v_k + \mu_1 w_1 + \dots + \mu_l w_l$. Die μ_i können nicht alle Null sein wegen der Annahme an u . O.B.d.A. sei $\mu_1 \neq 0$. Dann ist $\{v_1, \dots, v_k, u, w_2, \dots, w_l\}$ auch eine Basis.

Daher findet folgender Greedyalgorithmus immer eine Basis: Beginne mit beliebigem Vektor $v_1 \neq 0$. Sind $v_1, \dots, v_k$ schon gefunden, so wähle für v_{k+1} irgendeinen Vektor, der von $v_1, \dots, v_k$ linear unabhängig ist. Gibt es keinen solchen mehr, so liegt Basis vor.

Huffman-Codes

Definition:

Für Wörter $v, w \in \{0, 1\}^*$ heißt v ein **Präfix** von w , falls $w = vu$ für ein $u \in \{0, 1\}^*$.

Ein **Präfixcode** ist eine Menge von Wörtern $C \subseteq \{0, 1\}^*$ mit:
für alle $v, w \in C$ ist v nicht Präfix von w .

Das Huffman-Code-Problem:

Gegeben: Alphabet $A = \{a_1, \dots, a_n\}$ mit Häufigkeiten $h(a_i) \in \mathbb{R}_+$

Ziel: Finde Präfixcode $C = \{c_1, \dots, c_n\}$ derart dass $\sum_{i=1}^n h(a_i) \cdot |c_i|$ minimal wird.

Ein solches C heißt optimaler Präfixcode, oder **Huffman-Code**; diese werden bei der Datenkompression verwendet.

Der Huffman-Algorithmus

- Erzeuge Knoten $z_1, \dots, z_n$ mit Feldern $B[z_i] \leftarrow a_i$ und $H[z_i] \leftarrow h(a_i)$.
- Speichere diese in einer **Priority Queue** Q mit Operationen INSERT und EXTRACT-MIN (z.B. realisiert als Heap).
- Baue sukzessive einen binären Baum auf: Blätter sind die Knoten z_i , innere Knoten x haben nur ein Feld $H[x]$.
- Codewort c_i entspricht Pfad im Baum zu z_i , wobei $left \hat{=} 0$ und $right \hat{=} 1$.
- Aufbau des Baumes nach dem folgenden Algorithmus:
 - for** $i \leftarrow 1$ **to** $n - 1$ **do**
 - erzeuge neuen Knoten x
 - $left[x] \leftarrow \text{EXTRACT-MIN}(Q)$
 - $right[x] \leftarrow \text{EXTRACT-MIN}(Q)$
 - $H[x] \leftarrow H[left[x]] + H[right[x]]$
 - INSERT(Q, x)
 - return** EXTRACT-MIN(Q)
- Laufzeit: $O(n) + 3(n - 1) \cdot O(\log n) = O(n \log n)$.

Korrektheit des Huffman-Algorithmus

Wir identifizieren Huffman-Codes mit Code-Bäumen, wie sie der Algorithmus liefert. Also binäre Bäume, deren Knoten k mit Zahlen $H[k]$ beschriftet sind, so dass gilt:

- k innerer Knoten mit Kindern k_1 und k_2 : $H[k] = H[k_1] + H[k_2]$
- k Blatt: $H[k] = h(B[k])$.

Die Kosten solch eines Baumes sind

$$K(T) := \sum_{x \in A} \text{„Tiefe von } x \text{ in } T\text{“} \cdot h(x) = \sum_{k \text{ innerer Knoten von } T} H[k]$$

Dies deshalb, weil im rechtsstehenden Ausdruck jede relative Häufigkeit $h(x)$ gerade so oft summiert wird, wie der entsprechende Buchstabe tief steht.

Nehmen wir an, dass zu einem gewissen Zeitpunkt die Queue die Teilbäume $T_1, \dots, T_m$ enthalte mit Wurzelbeschriftungen $p_1, \dots, p_m$, wobei $p_1 \leq p_2 \leq p_j$ für $j > 2$. Nach Induktionsannahme existiert ein optimaler Codebaum T , der die T_i als Teilbäume besitzt. Sind in T die Wurzeln von T_1 und T_2 Geschwister, so erweitert T auch die Queue im nächsten Schritt zu optimaler Lösung. Anderenfalls seien k_1 und k_2 die Eltern von T_1, T_2 und T_{i_1}, T_{i_2} die entsprechenden Geschwister. Es gilt $H[k_1] = p_1 + p_{i_1}$, $H[k_2] = p_2 + p_{i_2}$. Sei d_1 die Tiefe von k_1 in T und d_2 die Tiefe von

Korrektheit des Huffman-Algorithmus

k_2 in T . O.B.d.A. sei $d_1 \geq d_2$. Sei T' der Baum, den man erhält, indem man T_{i_1} und T_2 vertauscht, sodass nunmehr k_1 die Bäume T_1, T_2 als Kinder hat und k_2 die Kinder T_{i_1} und T_{i_2} hat.

Es gilt $K[T] = d_1(p_1 + p_{i_1}) + d_2(p_2 + p_{i_2}) + c$, $K[T'] = d_1(p_1 + p_2) + d_2(p_{i_2} + p_{i_2}) + c$ für eine Konstante c . Es folgt $K[T'] \leq K[T]$, somit ist T' auch optimal und setzt die vom Huffman Algorithmus getroffene Entscheidung zu Optimallösung fort.

Dynamische Programmierung

Entwurfsprinzip, hauptsächlich für Algorithmen für Optimierungsprobleme:

- **Zerlegung** des Problems in kleinere, gleichartige Teilprobleme.
- Lösung ist zusammengesetzt aus den Lösungen der Teilprobleme.
- Aber: Teilprobleme sind nicht unabhängig, sondern haben ihrerseits **gemeinsame Teilprobleme**.

~> rekursives Divide-and-Conquer-Verfahren würde die gleichen Teilprobleme immer wieder lösen.

Lösung: Rekursion mit **Memoisierung**, oder besser:

Dynamische Programmierung:

Bottom-Up-Berechnung der Lösungen größerer Probleme aus denen kleinerer, die dabei in einer **Tabelle** gespeichert werden.

Einfaches Beispiel: Matrizen-Kettenmultiplikation

Problem: n Matrizen $M_1, \dots, M_n$ sollen multipliziert werden, wobei M_i eine $n_i \times n_{i+1}$ -Matrix ist. Aufwand hängt von der Klammerung ab.

Bezeichne $M_{i..j}$ das Produkt $M_i \cdot M_{i+1} \cdot \dots \cdot M_j$, und $m[i, j]$ die minimale Zahl von Multiplikationen zum Berechnen von $M_{i..j}$.

1. Optimale Klammerung von $M_{1..n}$ ist von der Form $M_{1..k} \cdot M_{k+1..n}$, für optimale Klammerungen von $M_{1..k}$ und $M_{k+1..n}$.
2. Deshalb gilt:

$$m[i, j] = \begin{cases} 0 & \text{falls } i = j \\ \min_{i \leq k < j} m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} & \text{sonst.} \end{cases} \quad (*)$$

3. Fülle eine Tabelle, in der unten die $m[i, i] = 0$ stehen, und der h -ten Ebene die Werte $m[i, j]$ mit $j - i = h$. An der Spitze steht schließlich $m[1, n]$.
4. Speichere in jedem Schritt auch einen Wert $s[i, j]$, nämlich dasjenige k , für das in (*) das Minimum angenommen wird.

Längste gemeinsame Teilfolge (lgT)

Definition: $Z = \langle z_1, z_2, \dots, z_m \rangle$ ist **Teilfolge** von $X = \langle x_1, x_2, \dots, x_n \rangle$, falls es Indizes $i_1 < i_2 < \dots < i_m$ gibt mit $z_j = x_{i_j}$ für $j \leq m$.

Problem: Gegeben $X = \langle x_1, x_2, \dots, x_m \rangle$ und $Y = \langle y_1, y_2, \dots, y_n \rangle$

Ziel: finde $Z = \langle z_1, z_2, \dots, z_k \rangle$ maximaler Länge k , das Teilfolge von X und Y ist.

Abkürzung: $X_i := \langle x_1, x_2, \dots, x_i \rangle$ ist das i -te Präfix von X .

Lösung mit dynamischer Programmierung:

1. Sei Z lgT von X und Y . Dann gilt:

(a) Ist $x_m = y_n$, so ist $z_k = x_m = y_n$, und Z_{k-1} ist lgT von X_{m-1} und Y_{n-1} .

(b) Ist $x_m \neq y_n$, so gilt:

- Ist $z_k \neq x_m$, dann ist Z lgT von X_{m-1} und Y .
- Ist $z_k \neq y_n$, dann ist Z lgT von X und Y_{n-1} .

Längste gemeinsame Teilfolge (lgT)

2. Sei $c[i, j]$ die Länge einer lgT von X_i und Y_j . Es gilt:

$$c[i, j] = \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0, \\ c[i - 1, j - 1] + 1 & \text{falls } i, j > 0 \text{ und } x_i = y_j \\ \max(c[i - 1, j], c[i, j - 1]) & \text{falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}$$

3. Fülle die Tabelle der $c[i, j]$ zeilenweise, jede Zeile von links nach rechts.

Am Ende erhält man $c[m, n]$, die Länge einer lgT von X und Y .

4. Für $i, j \geq 1$, speichere einen Wert $b[i, j] \in \{\uparrow, \swarrow, \leftarrow\}$, der anzeigt, wie $c[i, j]$ berechnet wurde:

$$\uparrow : c[i, j] = c[i - 1, j]$$

$$\swarrow : c[i, j] = c[i - 1, j - 1] + 1$$

$$\leftarrow : c[i, j] = c[i, j - 1]$$

Damit läßt sich eine lgT aus der Tabelle ablesen.

Zusammenfassung Dynamische Programmierung

- Ausgangspunkt ist immer eine rekursive Lösung des gegebenen Problems.
- Dynamische Programmierung bietet sich an, wenn die Zahl der rekursiven Aufrufe bei naiver Implementierung größer ist, als die Zahl der überhaupt möglichen voneinander verschiedenen Aufrufe der rekursive definierten Funktion.
- Man studiert dann die Reihenfolge, in der die Ergebnisse der Aufrufe benötigt werden und berechnet diese iterativ von unten her.

Amortisierungs-Analyse

Prinzip zur Analyse von Operationen auf Datenstrukturen.

- Betrachte nicht einzelne Operationen, sondern **Folgen** von Operationen.
- Statt der (*worst-case*-) Kosten jeder einzelnen Operation werden die **mittleren** Kosten der Operationen in der Folge bestimmt.
 $\rightsquigarrow$ teure Operationen können sich durch nachfolgende billigere **amortisieren**.
- Kosten einiger Operationen werden zu hoch, andere zu niedrig veranschlagt.
- Drei Methoden:
 1. **Aggregat-Methode:**
 Berechne worst-case Kosten $T(n)$ einer Folge von n Operationen.
 Amortisierte Kosten: $T(n)/n$.
 2. **Konto-Methode:**
 Elemente der Datenstruktur haben ein Konto, auf dem zuviel berechnete Kosten gutgeschrieben werden. Zu niedrig veranschlagte Operationen werden aus dem Konto bezahlt.
 3. **Potenzial-Methode:**
 Ähnlich der Konto-Methode, aber gesamte zuviel berechnete Kosten werden als *potenzielle Energie* der Datenstruktur aufgefasst.

Einfaches Beispiel: Binärzähler

Datenstruktur: Array A von $length[A] = k$ Bits.

Einzig Operation:

INCREMENT(A)

```
1  $i \leftarrow 0$ 
2 while  $i < length[A]$  und  $A[i] = 1$ 
3     do  $A[i] \leftarrow 0$ 
4          $i \leftarrow i + 1$ 
5 if  $i < length[A]$ 
6     then  $A[i] \leftarrow 1$ 
```

Worst-case-Kosten eines INCREMENT: $\lceil \log_2(n + 1) \rceil$ Wertzuweisungen.

Aggregat-Methode

Betrachte Folge von n INCREMENT-Operationen (Annahme: $k \geq \lceil \log_2(n+1) \rceil$).

Berechne Kosten $T(n)$ der ganzen Folge von n INCREMENT-Operationen:

- **Beobachtung:**

$A[0]$ wird bei jedem INCREMENT verändert,

$A[1]$ nur bei jedem zweiten,

$A[2]$ nur bei jedem vierten,

etc.

- Gesamtzahl der Wertzuweisungen bei n INCREMENT-Operationen:

$$T(n) \leq \sum_{i=0}^{\lceil \log_2(n+1) \rceil} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2n$$

- Also: $T(n) = O(n)$, somit amortisierte Kosten jedes INCREMENT:

$$O(n)/n = O(1)$$

Konto-Methode

Für die i -te Operation:

tatsächliche Kosten c_i amortisierte Kosten $\hat{c}_i$.

- Elemente der Datenstruktur haben Konten.
- Ist $\hat{c}_i > c_i$, so wird die Differenz $\hat{c}_i - c_i$ auf ein Konto gutgeschrieben.
- Ist $\hat{c}_i < c_i$, so muss die Differenz $c_i - \hat{c}_i$ aus einem Konto bezahlt werden.
 $\leadsto$ $\hat{c}_i$ müssen so gewählt sein, dass genügend Guthaben auf den Konten ist.

Im Beispiel:

- Kosten einer Wertzuweisung seien 1 EUR.
- Jede Array-Position $j \leq k$ hat ein Konto.
- Veranschlage amortisierte Kosten eines INCREMENT: 2 EUR.
- Jedes INCREMENT setzt nur ein Bit $A[i] \leftarrow 1$ (Zeile 6):
Zahle dafür 1 EUR, und schreibe 1EUR auf dessen Konto gut.
 $\leadsto$ Jedes j mit $A[j] = 1$ hat ein Guthaben von 1 EUR.
Zahle damit die Wertzuweisungen $A[i] \leftarrow 0$ in Zeile 3.

Potenzial-Methode

Sei D_i die Datenstruktur nach der i -ten Operation.

- Definiere eine **Potenzialfunktion** Φ , die jedem D_i ein $\Phi(D_i) \in \mathbb{R}$ zuordnet.
- Amortisierte Kosten $\hat{c}_i$ definiert durch $\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$.

- Gesamte amortisierte Kosten:

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$$

- Ist $\Phi(D_n) \geq \Phi(D_0)$, so sind tatsächliche Gesamtkosten höchstens die amortisierten Gesamtkosten.
- Meist ist n variabel, und man setzt $\Phi(D_0) = 0$, und $\Phi(D_i) \geq 0$ für $i > 0$.

Potenzial-Methode

Im Beispiel:

- Definiere $\Phi(D_i) = b_i :=$ Anzahl der 1 im Zähler nach i INCREMENT-Operationen .
- Damit ist $\Phi(D_0) = 0$, und $\Phi(D_i) > 0$ für alle $i > 0$.
- Sei t_i die Zahl der Durchläufe der **while** -Schleife beim i -ten INCREMENT.
- Damit ist $c_i = t_i + 1$: Es werden t_i bits auf 0 gesetzt, und eines auf 1.
- Ausserdem gilt: $b_i = b_{i-1} - t_i + 1$.
- Daher ist die Potenzialdifferenz:

$$\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} = (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i ,$$

also sind die amortisierten Kosten:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = t_i + 1 + (1 - t_i) = 2$$

Queue als zwei stacks

Eine Queue Q kann mittels zweier Keller S_{in} und S_{out} realisiert werden:

PUT(Q, x)

 PUSH(S_{in}, x)

GET(Q)

if EMPTY(S_{out})

then while nicht EMPTY(S_{in}) **do**

 PUSH($S_{\text{out}}, \text{POP}(S_{\text{in}})$)

return POP(S_{out})

Wieviele Kelleroperationen benötigen die Queue-Operationen im **worst-case**?

PUT(Q, x): 1

GET(Q): $2n + 1$

Amortisierungsanalyse der Queue

Definiere **Potenzial** $\Phi(Q) := 2 \cdot |S_{\text{in}}|$.

Sei Q' die Queue nach Anwendung der Operation auf Q .

PUT(Q, x): es ist $\Phi(Q') = \Phi(Q) + 2$.

$$\begin{aligned} c_{\text{PUT}} &= 1 & \hat{c}_{\text{PUT}} &= 1 + \Phi(Q') - \Phi(Q) \\ & & &= 1 + 2 = 3 \end{aligned}$$

GET(Q), Fall 1: S_{out} nicht leer. Dann ist $\Phi(Q') = \Phi(Q)$.

$$c_{\text{GET}} = 1 \qquad \hat{c}_{\text{GET}} = 1 + \Phi(Q') - \Phi(Q) = 1$$

GET(Q), Fall 2: S_{out} leer. Dann ist $\Phi(Q') = 0$ und $\Phi(Q) = 2 \cdot |S_{\text{in}}| = 2n$.

$$\begin{aligned} c_{\text{GET}} &= 2n + 1 & \hat{c}_{\text{GET}} &= 1 + \Phi(Q') - \Phi(Q) \\ & & &= 2n + 1 + 0 - 2n = 2 \end{aligned}$$

Union-Find: Datenstruktur für disjunkte Mengen

Es soll eine Familie von disjunkten dynamischen Mengen

$$M_1, \dots, M_k \quad \text{mit} \quad M_i \cap M_j = \emptyset \quad \text{für} \quad i \neq j$$

verwaltet werden.

Dabei sollen folgende Operationen zur Verfügung stehen:

- **MAKE-SET**(x) fügt die Menge $\{x\}$ zur Familie hinzu. Es ist Aufgabe des Benutzers, sicherzustellen dass x vorher in keiner der M_i vorkommt.
- **FIND-SET**(x) liefert ein **kanonisches Element** der Menge M_i mit $x \in M_i$, also $\text{FIND-SET}(x) = \text{FIND-SET}(y)$, falls x und y in der gleichen Menge M_i liegen.
- **UNION**(x, y) ersetzt die Mengen M_i und M_j mit $x \in M_i, y \in M_j$ durch ihre **Vereinigung** $M_i \cup M_j$.

Typische Anwendungen: Äquivalenzklassen einer Äquivalenzrelation, Zusammenhangskomponenten eines Graphen , ...

Union-Find: Realisierung als Wälder

Jede Menge wird durch einen Baum dargestellt. Knoten x haben Zeiger $p[x]$ auf ihren Vater. Kanonische Elemente sind an der Wurzel und haben $p[x] = x$.

Optimierungen:

- Rangfunktion: Jedes Element hat **Rang** $\approx \log(|\text{Teilbaum ab } x|)$
- **Pfadkompression**: Hänge jeden besuchten Knoten direkt an die Wurzel.

MAKE-SET(x)

$p[x] \leftarrow x$
 $rank[x] \leftarrow 0$

UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))

FIND-SET(x)

if $x \neq p[x]$
 then $p[x] \leftarrow \text{FIND-SET}(p[x])$
return $p[x]$

LINK(x, y)

if $rank[x] > rank[y]$
 then $p[y] \leftarrow x$
 else $p[x] \leftarrow y$
 if $rank[x] = rank[y]$
 then $rank[y] \leftarrow rank[y] + 1$

Analyse der *Union-Find*-Wälder

Sei $size(x)$ die Größe des Teilbaumes unter x .

Lemma: Für jede Wurzel x gilt: $size(x) \geq 2^{rank[x]}$

Lemma: Für jedes $r \geq 1$ gibt es höchstens $n/2^r$ Knoten x mit $rank[x] = r$.

Korollar: Jeder Knoten hat $rank[x] \leq \log_2 n$.

Lemma: Für jede Wurzel x ist die Höhe $h(x)$ des Baumes unter x höchstens $h(x) \leq rank[x]$.

Satz: Jede Folge von m MAKE-SET-, UNION- und FIND-SET-Operationen, von denen n MAKE-SET sind, hat für Wälder mit Rangfunktion eine Laufzeit von $O(m \log n)$.

Komplexitätsverbesserung durch Pfadkompression

Definition: Der iterierte Logarithmus $\log^* n$ ist gegeben durch:

$$\log^{(0)} n := n$$
$$\log^{(i+1)} n := \begin{cases} \log \log^{(i)} n & \text{falls } \log^{(i)} n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

$$\log^* n := \min\{i ; \log^{(i)} n \leq 1\}$$

Satz: Jede Folge von m MAKE-SET-, UNION- und FIND-SET-Operationen, von denen n MAKE-SET sind, hat für Wälder mit Rangfunktion und Pfadkompression eine Laufzeit von $O(m \log^* n)$.

Bemerkung: Für $n \leq 10^{19.728}$ ist $\log^* n \leq 5$.

Backtracking

Mögliche Lösungen erscheinen als Blätter eines (immensen) Baumes.

Teillösungen entsprechen den inneren Knoten.

Der Baum wird gemäß der Tiefensuche durchgearbeitet.

Sieht man einer Teillösung bereits an, dass sie nicht zu einer Lösung vervollständigt werden kann, so wird der entsprechende Teilbaum nicht weiter verfolgt.

Damenproblem

Man soll n Damen auf einem $n \times n$ Schachbrett so platzieren, dass keine Dame eine andere schlägt.

Wir platzieren die Damen zeilenweise von unten her und repräsentieren Teillösungen als Listen.

Beispiel: $[0;1;6]$ bedeutet:

Die unterste Zeile enthält eine Dame auf Position 6 (=siebtes Feld von links).

Die zweite Zeile von unten enthält eine Dame auf Position 1 (=zweites Feld von links).

Die dritte Zeile von unten enthält eine Dame auf Position 0 (=erstes Feld von links).

Solch eine Teillösung heißt **konsistent**, wenn keine Dame eine andere schlägt.

Testen auf Inkonsistenz

```
let inkonsistent i l =  
  List.mem i l ||  
  let rec test j = if j >= List.length l then false else  
    List.nth l j = i+j+1 || List.nth l j = i-j-1 || test (j+1)  
  in test 0
```

Ist l selbst konsistent, so ist $i::l$ konsistent genau dann, wenn `inkonsistent i l = false`.

Lösung mit Backtracking

```
let rec finde_damen l =
  if List.length l = anzahl_damen then l else
  let rec probiere i =
    if i >= anzahl_damen then [] else
    if not (inkonsistent i l) then
      let result = finde_damen (i::l) in
        if result = [] then probiere (i+1)
        else result
    else probiere (i+1)
  in probiere 0
```

Ist l konsistent, so liefert `finde_damen l` eine Erweiterung von l auf `anzahl_damen` Damen, falls es eine gibt. Es wird `[]` zurückgegeben, falls es keine gibt.

Eine Lösung: [3; 1; 6; 2; 5; 7; 4; 0]

Iterative Deepening

Oft ist die Problemstellung so, dass eine Lösung mit steigender Tiefe im Lösungsbaum immer unwahrscheinlicher oder weniger brauchbar wird.

Im Prinzip würde es sich dann anbieten, die Tiefensuche durch eine Breitensuche zu ersetzen, was allerdings meist zu aufwendig ist (Speicherplatzbedarf steigt exponentiell mit der Tiefe).

Man kann dann der Reihe nach Tiefensuche bis maximale Tiefe 1, dann bis zur maximalen Tiefe 2, dann 3, dann 4, etc. durchführen.

Zwar macht man dann beim nächsten Tiefenschritt die gesamte bisher geleistete Arbeit nochmals; dennoch erzielt man so eine höhere Effizienz als bei der Breitensuche. Diese Technik heißt *iterative deepening*.

Branch and bound

Oft sind die Lösungen eines Baum-Suchproblems angeordnet gemäß ihrer Güte und man sucht entweder die beste Lösung, oder aber eine deren Güte eine bestimmte Schranke überschreitet.

Manchmal kann man die Güte von Lösungen schon anhand einer Teillösung abschätzen. Zeigt sich aufgrund dieser Abschätzung, dass keine Lösung in einem Teilbaum besser ist, als eine bereits gefundene, so braucht man diesen Teilbaum nicht weiter zu verfolgen.

Diese Verbesserung von Backtracking bezeichnet man als *Branch-and-Bound*.

Abstraktes Beispiel

Sei $f : \text{bool list} \rightarrow \text{int}$ eine Funktion mit der Eigenschaft, dass $f(l) \geq f(l@l')$ und $f(l) \geq 0$.

Man bestimme $\max\{f(l) : |l| = n\}$.

Folgendes Programm kann das tun:

```
let rec finde_optimum f n =
  let rec finde_erweiterung teilloesung bisherbestor tiefe =
    if bisherbestor >= f(teilloesung) then bisherbestor else
    if tiefe = 0 then f(teilloesung) else
    let optimum_links = finde_erweiterung (teilloesung@[false])
      bisherbestor (tiefe-1) in
    let optimum_rechts = finde_erweiterung (teilloesung@[true])
      optimum_links (tiefe-1) in
    optimum_rechts
  in finde_erweiterung [] 0 n
```

In der Praxis verwendet man B&B gern bei der linearen Optimierung mit Ganzzahligkeitsbedingungen (ILP).